ISSUE 15 - AUG 2013

Get printed copies at themagpi.com

The MagPi

A Magazine for Raspberry Pi Users

Old-school gaming

Arduino programming
Power extension
Camera module
Bare metal
Assembler
Python

ISSN 2051-9990

Raspberry Pi is a trademark of The Raspberry Pi Foundation.
This magazine was created using a Raspberry Pi computer.

CC BY NC SA

The MagPi

http://www.themagpi.com

Welcome to the 15th issue of The MagPi.

Are you bored of school holidays or just want some down time after a hard day at work? This month's issue has something for everyone! We take a look at the 'Multiple Arcade Machine Emulator' and reflect back at some of the arcade history's greatest games, describing how you can turn your Pi into a retro gaming console!

If that's not enough, we delve deeper into the the partnership made in heaven, The Raspberry Pi - Arduino double act. We look at connecting the two and even the possibility of controlling your Arduino from the command line+

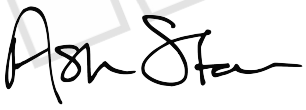
James Hughes discusses advanced usage of the camera module and we publish more on Cocktail MegaPower and Pi Matrix, where Bruce Hall describes how to produce lighting routines for this clever piece of kit+

We are excited to start you on an epic journey towards making your own operating system in the first in a series by Martin Kalitis titled 'Bake your own Pi filling +

We are proud to introduce yet another language to our readers, XML, along with more from favourites Assembler and Python+

On top of all this, as always, we keep you up to date with Raspberry Pi events across the world. Phew! That's a lot to get your teeth into. We better get started. Enjoy!

Ash Stone



Chief Editor of The MagPi

The MagPi Team

Ash Stone - Chief Editor / Administration / Layout
W.H. Bell - Issue Editor / Layout / Graphics / Administration
Bryan Butler - Page Design / Graphics
Ian McAlpine - Layout / Tester
Chris 'tzj' Stagg - Tester
Colin Deady - Layout
Matt Judge - Website / Administration
Aaron Shaw - Layout
Shelton Caruthers - Proof Reading
Meltwater - Proof Reading
James Nelson - Tester, Proof Reading
Sai Yamanoor - Tester
Claire Price - Layout
Phil Tesseyman - Tester
Steve Drew - Layout
Courtney Blush - Proof Reading
Amy-Clare Martin - Proof Reading

Contents

4 **USB ADRDUINO LINK**
Using Nanpy to connect your Raspberry Pi to An Arduino

8 **COMMAND LINE ARDUINO PROGRAMMING**
Using the Ino command line toolkit

10 **THE RASPBERRY PI CAMERA MODULE**
Part 2: Advanced Operation

14 **A COCKTAIL OF EXPANSION BOARDS**
Part 4: MegaPower: DC-DC converter and an ATmega328 MCU

16 **THE RASCLOCK**
Raspberry Pi timekeeping with a real time clock

18 **PI MATRIX**
Part 3: Building a toolkit of patterns

20 **MAME - MULTIPLE ARCADE MACHINE EMULATOR**
Play historic games on the Raspberry Pi

24 **<XML />**
Part 1: an introduction to XML

28 **MY OS: BUILD A CUSTOMISED OPERATING SYSTEM**
Part 1: Bake your own Pi filling

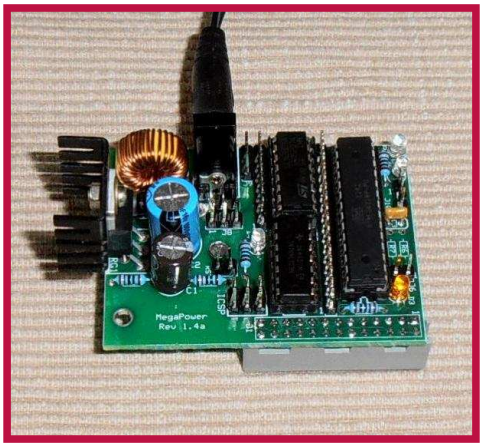
30 **ASSEMBLY PROGRAMMING WITH RISC OS**
Part 2: Low-level coding

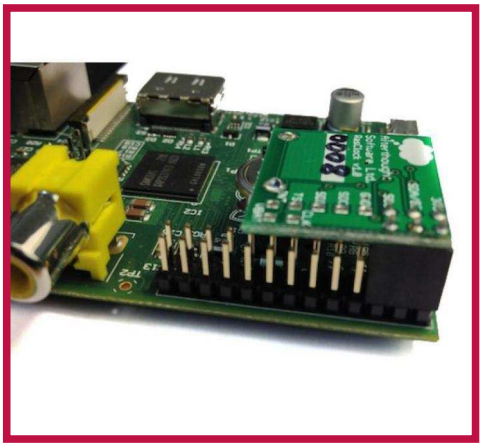
33 **THIS MONTH'S EVENTS GUIDE**
Preston, Manchester, Powys, Gateshead

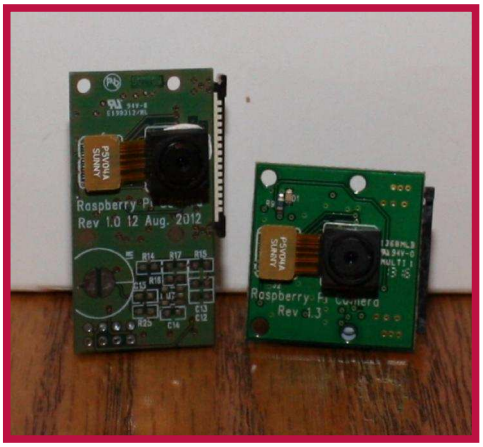
35 **BOOK REVIEW**
Charm Programming on the Raspberry Pi

36 **THE PYTHON PIT**
An introduction to Python iterators and generators

40 **FEEDBACK**
Have your say about The MagPi

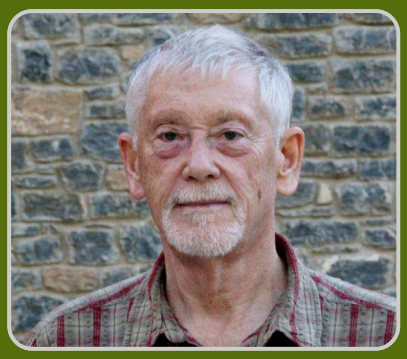






Using Nanpy to connect your Raspberry Pi to an Arduino

DIFFICULTY : INTERMEDIATE



Tony Goodhew

Guest Writer

If you have been using your Raspberry Pi with LEDs and switches from Python and wish to progress to the next level (reading analogue values and adjusting output voltages with PWM) then using an Arduino as a cheap and expandable input/output board is a good option, while still using Python. (You also gain a board that can be used on its own to control robots which you program from your Raspberry Pi). This uses a very safe USB connection allowing the two computers to communicate in both directions. The Arduino provides an extra 14 digital pins (0 – 13), six of which have PWM facilities and six analogue pins (10 bit, range 0-1023, A0-A5) which can also be used as digital I/O pins. An Arduino Uno R3 costs less than your Raspberry Pi and you do not have any soldering to do!

Preparing the SD card

You will need a 4 GB class 4 card (slow). Copy the latest version of Raspbian Wheezy onto it and expand the root partition. Then reboot the Raspberry Pi.

Install setuptools

You need python setuptools to install nanpy on your card. This is not in the current distribution, however you can download it from the web.

Start the Midori web browser and type in the URL box https://pypi.python.org/pypi/setuptools

Scroll down to the Linux instructions and then on to the downloads. We want the file:

```
setuptools-0.6c11-py2.7.egg
```

Click on it and you will be asked to open or download. Click on SAVE. It downloads very quickly. Once the download has finished, close Midori and you should see the egg file in the pi directory.

Next, open the LX Terminal and type in:

```
sudo sh setuptools-0.6c11-py2.7.egg
```

This is a very quick installation.

Install serialpy

Again, using the Midori web browser go to https://pypi.python.org/pypi/pyserial. This time you want to download pyserial-2.6.tar.gz.

However, this time you want to make a temp folder and move the downloaded file into it - using LXTerminal:

```
cd temp      # Change to the temp directory
gunzip pyserial-2.6.tar.gz      # to unzip it
tar -xvf pyserial-2.6.tar       # to untar it
cd pyserial-2.6    # move into the new folder
sudo python setup.py install   # to install it
```

The Raspberry Pi can now use serial communication.

Install the Arduino software

To begin with, type 'startx' to start the GUI (unless you set it to boot into the GUI automatically). Then open the LX Terminal and type in the the following commands:

```
sudo apt-get update
sudo apt-get install arduino
```

Answer 'Y' when asked if you want to continue. This installation takes some time...

You can now program your Arduino from the Arduino IDE.

Install Nanpy

The next step is to download the nanpy files. You could use wget, but I find it much easier to do on a Windows PC and then transfer the unzipped folder to the Raspberry Pi via a memory stick. On a PC, using your web browser, navigate to the git repository https://github.com/nanpy/nanpy and click on the ZIP button. This downloads the zipped directory to your computer. Unzip it and copy the nanpy-master directory via a USB stick to your pi directory.

Next, connect your Arduino via a USB cable to the Raspberry Pi. Then open the LX Terminal and navigate to the firmware directory in nanpy-master:

```
cd nanpy-master
cd firmware
export BOARD=uno
# (Type 'make boards' for a full list)
make
make upload   # This also takes some time.....
```

This RED section needs to be done each time you connect the Arduino to the Raspberry Pi. It loads the Arduino part of nanpy into the Arduino. Now type:

```
cd .. # Move back to nanpy-master directory
sudo python setup.py install
```

This adds the Raspberry Pi part of nanpy to Python2 and only needs to be done once. You can now use the Arduino as an I/O board for the Raspberry Pi. This adds 20 extra I/O pins – 6 can be used for 10-bit analog inputs (range 0-1023) and 6 as PWM outputs (range 0-255).

Testing

The Arduino really needs to be connected via a powered USB hub so that it does not take too much power from the Raspberry Pi.

On your Arduino connect an LED in series with a resistor (about 250 Ohm for protection) between pin 10 and ground. The longer lead of the LED, the anode, goes towards pin10 and the shorter, the cathode, towards GND. Pin10 allows PWM (Pulse Width Modulation).

From the LXDE GUI on your Raspberry Pi, start LXTerminal once again and type:

```
sudo idle
```

With sudo you can run the program from the Run menu in IDLE. Click on File, then New window (to open a new window) and type in the following program:

```
#!/usr/bin/env python
# LED with 250 Ohm resistor on Pin 10 to GND
# Tony Goodhew - 10 May 2013
from nanpy import Arduino
from nanpy import serial_manager
serial_manager.connect('/dev/ttyACM0')         # serial  connection to Arduino
from time import sleep

LED =10                          # LED on Arduino Pin 10 (with PWM)
Arduino.pinMode(LED, Arduino.OUTPUT)

print"Starting"
print"5 blinks"
for i in range(0,5):
    Arduino.digitalWrite(LED, Arduino.HIGH)
```

```
    sleep(0.5)
    Arduino.digitalWrite(LED, Arduino.LOW)
    sleep(0.5)

print"Changing brightness of LED"
bright = 128                          # Mid brightness
Arduino.analogWrite(LED, bright)
Arduino.digitalWrite(LED,Arduino.HIGH)       # Turn on LED

for i in range(0,200):
    bright = bright + 8
    if (bright > 200):    # LED already full on at this point
        bright = 0 # Minimum power to LED
    Arduino.analogWrite(LED, bright)   # Change PWM setting/brightness
    sleep(0.05)

Arduino.digitalWrite(LED,Arduino.LOW)        # Turn off LED
print"Finished"
```

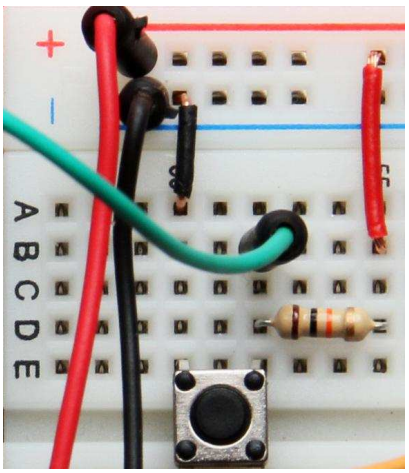
Save and run the program from the IDLE menu.

Problem: If you pull the USB cable out of the Arduino while the Pi is controlling it you may need to re-boot the Raspberry Pi before it will re-connect. You may also need to re-do the instructions in the 'Install Nanpy' section if the firmware gets corrupted.

Reading a single switch

Digital pins with switches need to be held HIGH (at 5V) or LOW (at 0V) until a switch is closed. When the user closes a switch the voltage of the pin changes to the opposite state - HIGH to LOW or LOW to HIGH. A 10K Ohm resistor is used to 'pull' the pin either HIGH or LOW. This is exactly the same as putting switches directly onto the Raspberry Pi's GPIO pins.

The image below shows the circuit on a breadboard. The wires connect to the Arduino: Red to 5V, black to GND and the green wire to pin8. The following code demonstrates how to read the switch value in Python.




```
#!/usr/bin/env python
# Button switch on pin 8
#       with 10K ohm pull up resistor

from nanpy import Arduino
from nanpy import serial_manager
serial_manager.connect('/dev/ttyACM0')

button = 8          # Switch on pin 8
count = 0           # Initialise counter
# Set Button pin for input
Arduino.pinMode(button, Arduino.INPUT)

print "Press the button 3 times"

while (count < 3):
    sw1 = Arduino.digitalRead(button)
    #Wait until switch is pressed
    while (sw1 == 1):
        sw1 = Arduino.digitalRead(button)
    count = count + 1
    print count
    #Wait until switch is released
    while (sw1 == 0):
        sw1 = Arduino.digitalRead(button)

print "Finished"
```

Thanks to Andrea Stagi for the software. See The MagPi, issue 8, page 12 for more details.

The next article will cover reading analogue ports and driving a liquid crystal display (LCD) so make sure to come back for more Nanpy goodness!

PiGlow

Raspberry Pi® Colour LED Plate

18-channel 8-bit PWM (0-255)
Individually addressable
6 hues + white
~300-500mcd per LED
Fits nicely inside a Pibow

http://shop.pimoroni.com/products/piglow

PIMORONI

TIMBER

The neat little log cabin for your Raspberry Pi®

Made from real
spruce hardwood

Available From:
http://pibow.com/

PIMORONI

Command line Arduino programming

DIFFICULTY : INTERMEDIATE

Nathan Bookham

Guest Writer

When I started getting into programming the Arduino, I stumbled across a great little tool that allows you to program your Arduino from the command line.

Ino (inotool.org) is a tool written in Python that allows you to easily compile an Arduino sketch without using a GUI (graphical user interface) or messing around with makefiles. It runs on Linux (as well as Mac, with Windows support coming soon) which means that we can run it on the Raspberry Pi.

Installing Ino

Ino is easy to install. Providing that you are running the latest version of Raspbian and follow the instructions, you shouldn't run into problems.

First of all, run the command below to update the apt repositories and to install any updates:

```
sudo apt-get -y update
sudo apt-get -y upgrade
```

After running this command, we need to install dependencies. Run the following command:

```
sudo apt-get install arduino picocom
python-setuptools
```

After the dependencies have been installed, use Python's easy_install command to install pip:

```
sudo easy_install-2.7 pip
```

Now we can use pip to install Ino and any Python dependencies we need. To do this we run the following command:

```
sudo pip install ino
```

Ino will now search the Python Package Index and download the files it requires. It will automatically install these packages as well.

To check that Ino has been installed we need to run a command to check that it can find the libraries and hardware files:

```
ino list-models
```

If a list of Arduino boards appear, you have successfully installed it. If you can't see any boards, make sure that the Arduino package is installed and up-to-date.

Using Ino

Ino is well documented - so if you get stuck have a look at the quick start guide online at

http://inotool.org/quickstart or run ino --help.

To create a new sketch make a new folder using mkdir:

```
mkdir Blink
```

Then open the Blink directory and use the Ino command to create the files and directories to create a sketch:

```
cd Blink
ino init
```

This creates two directories, src and lib. You place your .ino files in the src folder, and any libraries that you reference to in the lib folder. When Ino initializes the src directory, it automatically creates a file called sketch.ino.

Once everything has been initialised, you can create an .ino file and use your favorite text editor to write your sketch. You can use nano, vim, or any other text editor. Remember to place libraries in the lib folder, not the src folder.

Once you have created a sketch, we need to build it. If you use a board other than the Uno, be sure to change the board model. The Ino quickstart page shows you how to do this.
We need to change directory to the Blink folder, and then build it:

```
cd ..
ino build
```

Just wait a few moments while it compiles your sketch. Once done, you can upload it to your Arduino board. You can do this with the upload command:

```
ino upload
```

Ino is clever and can automatically detect which serial port your Arduino is on.
Once uploaded, that's it!

To view the serial output of your Arduino, just use the following command:

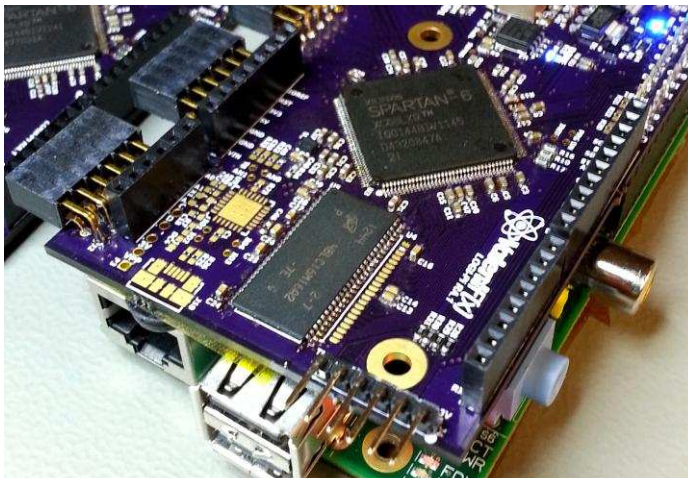
```
ino serial
```

Other expansion boards

Connecting an Arduino to a Raspberry Pi opens up a range of possibilities, with a large range of possible Arduino shields that can be added to a setup. There is a list of some of the boards at http://playground.arduino.cc/Main/HomePage

The list of boards available that can be directly connected to the Raspberry Pi is also increasing. For a summary of some of the possible extension boards, take at look at
http://elinux.org/RPi_Expansion_Boards
There are cards to allow the Raspberry Pi to be connected to batteries instead of the mains, several different LED boards, buzzers, and motor controllers. There are expansion boards to add other ports, such as RS232 or single wire interfaces. There are servo controllers and robotics boards. There is also a long list of multifunction boards, allowing several different signals to be read or written.

This month a FPGA (field programmable gate array) development board arrived in the post from ValentFx. The board needs some additional developers to bring its software up to speed. Please get in touch with the editor if you would be willing to help.



The Raspberry Pi camera - part 2

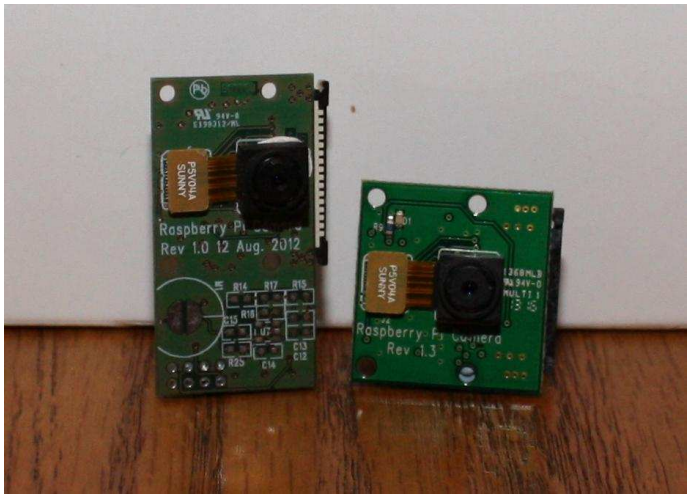
DIFFICULTY : BEGINNER

James Hughes

Guest Writer

Welcome back to part two of The MagPi mini series covering the fantastic Raspberry Pi camera module.

In part one we covered setting up your camera and it's basic operation. In this issue we begin to introduce you to some of the advanced features that the module is capable of, allowing you to capture images like a professional.



Sizing options

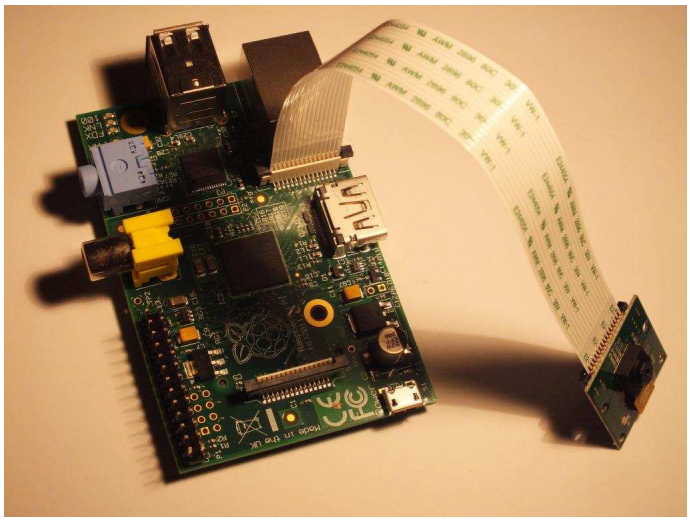
Lets begin by looking at sizing options. You can easily specify the size of your captures using the -w (--width) and -h (--height) options. These do exactly what you might expect - they change the resolution of the resultant captures (either stills or video). But you can also change the quality of the JPG stills using -q (--quality) and the quality of the H264 encoding using -b (--bitrate). Both these formats use what is known as lossy compression, that is they throw away detail in order to compress the files. The more detail that is thrown away, the smaller the file, it's simply a tradeoff between quality and file size.

Some examples..

```
raspistill -w 640 -h 480 -q 10 -o smallpic.jpg
```

```
raspivid -w 640 -h 480 -b 500000 -o \
smallvid.h264
```

The bitrate option is measured in bits per second. So in the example above, 500000 is 0.5Mbits/s. The is quite a low bitrate for 1080p video, but good enough for the small image size requested here. Full HD recording (the default) is supported, which is 1920x1080 at 30 frames per second (abbreviated to 1080p30), this requires a higher bitrate to avoid odd compression artefacts. It's well worth trying out various bitrates just to see how it affects the image quality. It's sometimes very surprising how low you can go before the video becomes unwatchable!



```
raspivid -t 10000 -b 1000000 -o \
highcompression.h264
```

In fact, the raspistill application allows us to save in 4 different file formats, although JPG is by far the fastest (and most common). The other formats supported are PNG, GIF and BMP. These are all lossless formats (no data is thrown away in order to decrease file size), and the file sizes are therefor much larger than JPG. They also take longer to save as there is no dedicated hardware acceleration in the GPU for them. You can select the type of file to output using the -e (--encoding) options.

```
raspistill -t 1000 -e png -o image.png
```

Timelapse mode

The raspistill app has a very useful feature called timelapse mode. Instead of just taking one picture, it takes sequence of pictures at a specified interval (use - tl) , until the timeout limit (-t) is reached. So, to take a picture every 5 seconds over a period of 50 seconds, use the following.

```
raspistill -t 50000 -tl 5000 -o image%d.jpg
```

The -t and -tl options are fairly obvious - but what's that odd filename specification? Well, if you just specified something like

```
raspistill -t 50000 -tl 5000 -o image.jpg
```

then all you pictures would be saved the the same filename, image.jpg. So you end up with just one picture! By adding a %d in your filename specifier, the image number is added to the filename at the place where the %d is. So our first example above would save a set of file as image1.jpg, image2.jpg...image10.jpg. In fact, the filename is generated uses the same formatting as the C language printf statement, so you can do something like this..

```
raspistill -t 50000 -tl 5000 -o image%04d.jpg
```

which specifies the number will be formated as 4 digits long and use preceding 0's to pad out those 4 digits, resulting in image0001.jpg, image0002.jpg...image0010.jpg being saved. See the printf format specifier documentation (man printf on a console command line) for more information.

Changing image parameters

Just like a compact camera, there are lots of options that can be applied to the images to change their effect, or the way they were taken. The following options are equally applicable to stills or video mode. Also note that some options may not be fully implemented at this stage, and you may see no effect when using them. This is because the applications were written to the full Camera API (application programming interface) available, not necessarily what was actually implemented under that API. I'm only going to describe working features here. Feel free to try all the options to see the effect they provide, and whether they actually do anything!

Firstly, we have the basic image operations. These are sharpness (-sh [-100 to 100]), contrast (-co [-100 to 100]), brightness (-br [0 to 100]) and saturation (-sa [-100 to 100]). All these settings are commonly found on cameras or even LCD televisions. Try them out with different numbers to see what effect they have - here's a start point.

```
raspistill -t 5000 -sh 100 -co 50 -br 25 \
-sa 50 -o image.jpg
```

Image effects (-ifx) are quite fun. These apply interesting filters to the image, like negative or emboss. Only one effect can be applied at a time, and the full set of effects available are :

```
negative, solarise, sketch, denoise, emboss,
oilpaint, hatch, gpen, pastel, watercolour,
film, blur saturation, colourswap, washedout,
posterise, colourpoint, colourbalance and
cartoon.
```

```
raspistill -t 5000 -ifx negative -o image.jpg
```

The colour effects option (-colfx) is interesting. Internally, the image is represented using a YUV colour space. YUV represents colour using the luminance, Y, and blue-luminance and red-luminance differences, UV. The colour effects option allows up to specify the values we are going to use for U and V. This gives us a quick and easy way to do black and white images, we just need to set UV equally to the middle of the range, which is 128. So

```
raspistill -t 5000 --colfx 128:128 \
-o image.jpg
```

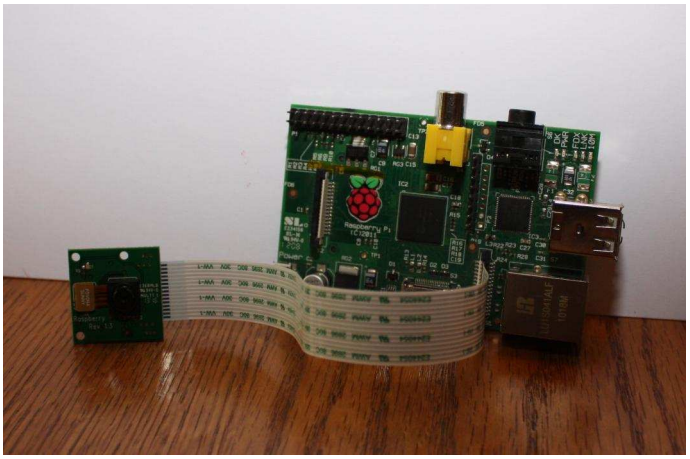
Other values for U and V give varying blue and red differences from the middle. Try some numbers!

The final image options I'm going to talk about are more about how the picture is taken rather than what processes are applied to the image. These are metering mode (-mm) and awb mode (-awb).

Metering mode specifies what area of the incoming image is used for determining the brightness of the image. Internally, there is a target brightness that is required, and the camera system adjusts the internal gain to hit that target. But to do that it need to know what brightness the incoming image is so it can be boosted to the required level. The area of the image that is used to determine that incoming brightness is what is defined by the metering mode. There are two usefully distinct options: average and spot. Spot takes the very centre of the image and uses that, average uses the whole image. So if you have a scene with a very bright point in the centre, using spot will almost certainly underexpose, so you should use average in that case.

```
raspistill -t 5000 -mm average -o image.jpg
```

AWB stands for automatic white balance. This is a complicated subject, but in very simple terms, it's the adjustment made to the image to



compensate for different lighting conditions to make whites look white. For example, different types of office light produce different lighting conditions, and the system needs to compensate for those conditions so the white walls still look white in the photograph. By default the awb selection is done automatically uses Bayesian analysis of the scene to make an educated guess on the lighting conditions. However, you can specify the AWB approach being used depending on the scene being captured. The options are :

```
 auto, sun, cloud, shade, tungsten,
fluorescent, incandescent, flash, horizon.
```

So to set up the AWB for a room lit by tungsten filament bulbs :

```
raspistill -t 5000 -awb tungsten -o image.jpg
```

Anything else?

We've covered many of the features available on the Raspberry Pi camera, but the best way to find out about them is to play with the camera, adjust settings and see what happens. To help with this there is a demo mode available which runs through a lot of the options automatically. The following example runs for 1 minute, changing an effect every 500ms (1/2second)

```
raspistill -d 500 -t 60000
```

One option not mentioned is the ability to output the image or video stream to stdout, so it can be piped to other applications (for example, network streaming). To do this, instead of specifying a filename for the -o option, you use the - (hyphen). The following example outputs the jpg data to stdout, where it is passed on the the file image.jpg. The effect is the same as specifying -o image.jpg.

```
raspistill -o - > image.jpg
```

Final words

As with everything Raspberry Pi, the camera is meant to be a learning experience. Many options are available for you to try out, some might be useful, some not so useful. Try them out, experiment, you cannot break it using any of the options!

The source code for the applications is publicly available, and was written in a very straightforward fashion (in C) with lots of comments to make it easier for people to modify it to their own purposes. Feel free to pile in, change stuff, see what it does. If you add a great new feature, make sure you post about it on the Raspberry Pi forums - you never know, it might make it in to the official applications!

MEGA POWER
Adding an ATmega MCU

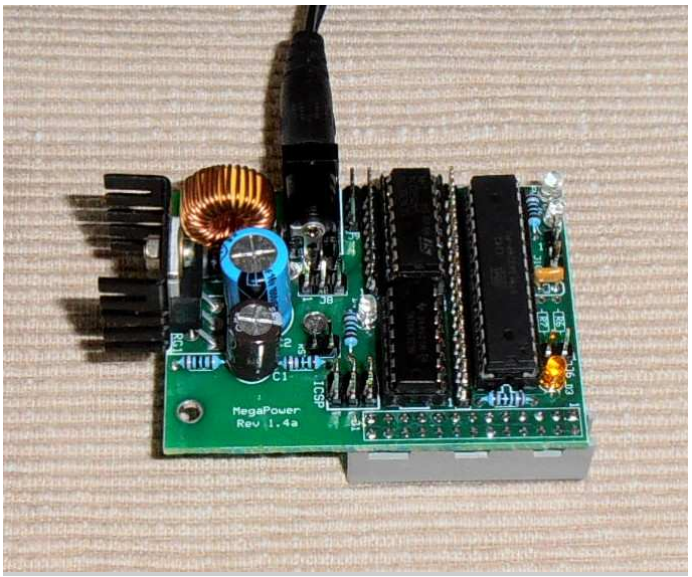
Lloyd Seaton

Guest Writer

This constructional project is for hobbyists who are confident with a soldering iron, who like to have options and are prepared to purchase their own components

Raspberry Pi continues to set new standards for affordability, accessibility, power and versatility but when the need arises to interface with "real world" digital and analog signals, the Arduino microcontroller platform offers significant advantages. This project involves the construction of a printed circuit assembly (MegaPower) that can be used to expand the Raspberry Pi's capabilities by inclusion of a companion microcontroller, an ATmega328 MCU. MegaPower can also include a buck DC-DC converter circuit to derive a 5V power supply from a higher DC voltage for powering of both MegaPower and Raspberry Pi. Alternatively, MegaPower can be used in standalone mode, connecting to Raspberry Pi only for initial programming of an Arduino sketch. As a constructor, the choices are yours to make!

The DC-DC buck converter components have been omitted from the MegaPower unit (opposite) as an economy measure but the ATmega MCU provides plenty of digital I/O and analog inputs for use by the Raspberry Pi and there are 7 Darlington outputs for driving relays etc. Unlike Gertboard's MCU, the MegaPower MCU operates at 5V for ease of interfacing with a wide range of conventional logic circuits. Level shifting circuitry provides permanent connections of SPI bus and UART to Raspberry Pi so that no jumpers are required for programming of Arduino sketches or for intercommunication using the Firmata protocol. Instead of mounting MegaPower on the Raspberry Pi's P1 connector via the grey 3M socket, the constructor may prefer to connect via ribbon cable, in which case header pins can be fitted to the upper side of the PCB instead of the 3M socket underneath.



The MegaPower unit pictured above is complete and being tested in standalone operation. The heat sink (far left) is needed if MegaPower is providing power for a Raspberry Pi Model B but should not normally be required by a Raspberry Pi Model A or for standalone operation. Near the right edge of the PCB there is provision for 2 pull-up resistors (R6 & R7) that may be required by

the I2C bus in standalone operation but should not be fitted if operation with a Raspberry Pi is intended. There are 4 LEDs, 2 of which indicate activity of the UART and SPI bus respectively. The other 2 LEDs are uncommitted and are available for general use by the programmed Arduino sketch.

Construction Of MegaPower

Before commencing construction of MegaPower it is important that you take the time to familiarise yourself with the design. By doing so you will maximise the educational value of your project and increase the likelihood that you will make the right configuration choices for your particular needs. A comprehensive design description document is available via the information blog at picocktails.blogspot.com (Issue 15 page). This document includes a schematic diagram, interface pinouts, a schedule of suggested components and suppliers and other helpful information. For constructors who are reasonably proficient, the project is relatively straight forward so long as it is approached with a sensible amount of planning and care, particularly with regard to component orientation. When in doubt, check it out!

Programming the ATmega MCU

The MCU operates from its internal clock at 8MHz or 1MHz so it is not immediately compatible with the Arduino IDE on Raspbian. However, compatibility is easily achieved.

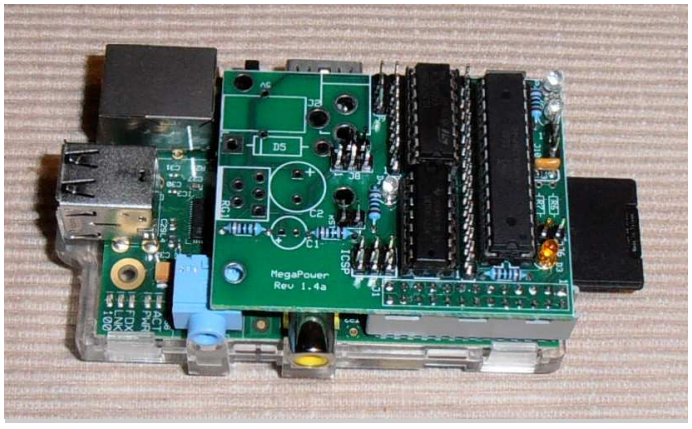
- Firstly, it is necessary to install Gordon Henderson's Arduino IDE extensions for Gertboard (projects.drogon.net/raspberry-pi/gertboard) which include the necessary support for programming an ATmega via the SPI bus and Raspberry Pi's GPIO ports.
- Secondly, you need to install the picocktails extensions for the Arduino IDE by following the procedure that is given on the Arduino page of the blog at picocktails.blogspot.com .

If you have already installed either or both of the above mentioned IDE extensions, you should not install them a second time or the consequences may be ugly. Having installed the IDE extensions you can activate the Arduino IDE and select: Tools -> Programmer -> Raspberri Pi GPIO. Then select: Tools -> Board -> MegaPower w/ ATmega328P (8MHz internal clock). Activate: Tools -> Burn Bootloader to initialise the ATmega fuses.

If you succeeded in burning the bootloader without error, it is very likely that you'll be able to successfully program an Arduino sketch. Try programming the Blink sketch from File -> Examples -> 01.Basics -> Blink but you'll need to edit the sketch to use Pin 12 instead of Pin 13 for controlling the LED. Then use: File -> Upload Using Programmer to send the modified Blink sketch to the MCU. After a surprisingly long delay ... the LED D4 should begin blinking. You can find further instructions on the (Issue 15 page) of picocktails.blogspot.com for programming the StandardFirmata sketch and running the pyfirmataManager.py test program.

Availability of PCBs

A fundamental goal of this series of articles is to help constructors to acquire superior PCBs at affordable prices so that they can better pursue their interest in applying the unique capabilities of the Raspberry Pi. Happily, there is now another option when seeking PCBs for projects such as MegaPower. The new WWW store, pi-supply.com is planning to supply PCBs for projects of this "cocktail series" (including MegaPower) and other projects in the near future. PCBs for MegaPower (plus MegaMini and MegaWire projects) will hopefully be available by the time you read this.



Raspberry Pi timekeeping with a real time clock

DIFFICULTY : BEGINNER

Jacob Marsh

ModMyPi

In order to achieve its miniature size and low price tag, several non-essential items usually found on a desktop computer had to be omitted from the Raspberry Pi. Laptops and computers keep time when the power is off by using a pre-installed, battery powered 'Real Time Clock' (RTC). An RTC module is not included with the Raspberry Pi, which instead updates the date and time automatically over the internet via Ethernet or WiFi. Subsequently, your Pi will revert back to the standard date and time settings when the network connection is removed. For projects which have no internet connection, you may want to add a low cost battery powered RTC to help your Pi keep time!

The RasClock has been specifically designed for use with the Raspberry Pi and plugs directly in to the Raspberry Pi's GPIO Ports. This article will walk you through its installation!

Step 1 - plug it in!

To avoid any damage to the module, make sure your Raspberry Pi is switched off and the RTC battery is firmly seated before installation. Plug the coin battery into the RTC by matching the positive on the battery with the positive on the module and then plug the RTC into the Raspberry Pi's GPIO pins. It sits on the 6 GPIO pins at the SD card end of the Raspberry Pi.

Step 2 - set-up

This RTC module is designed to be used in Raspbian. So the first step is to make sure you have the latest Raspbian Operating System (OS) installed on your Raspberry Pi (http://www.raspberrypi.org/downloads).
Currently the module requires the installation of a driver that is not included in the standard Raspbian distribution; however a pre-compiled installation package is available which makes setup nice and easy.

Make sure your Pi has internet access and grab the installation package off the internet from an LXTerminal window:

```
wget
http://afterthoughtsoftware.com/files/linux-
image-3.6.11-atsw-rtc_1.0_armhf.deb
```

(The wget command allows you to grab a file off the internet by providing a URL).

```
sudo dpkg -i linux-image-3.6.11-atsw-
rtc_1.0_armhf.deb
```

(The dpkg command enables the management

of Debian packages. The -i installs the package, or upgrades it if it is already installed).

This may take a couple of minutes to complete.

```
sudo cp /boot/vmlinuz-3.6.11-atsw-rtc+
/boot/kernel.img
```

(The cp command stands for copy. Here, we need to copy the RTC module's boot file to the Raspberry Pi boot directory).

The next step involves editing the text in the Raspberry Pi boot files. I usually use nano text editor for these minor changes - it's basic, pre-installed and easy to master. System commands for nano are enabled by holding the CTRL key (denoted as ^ in nano) whilst pressing the relevant command e.g. CTRL+X to exit.

We need to configure Raspbian to load the RTC drivers at boot by adding the boot information to the /etc/modules configuration file:

```
sudo nano /etc/modules
```

(This will open the 'modules' file within nano text editor and allow you to make changes. To add text simply use the arrows keys to browse to the next line in the boot file and add the following text, one per line. Then exit nano (CTRL+X) and don't forget to save those changes!

```
i2c-bcm2708
rtc-pcf2127a
```

The final step in set-up is to register the RTC module when the Raspberry Pi boots and set the system clock from the RTC. When editing files always follow the instructions outlined at the top of the file denoted by #. For example, the file we are just about to edit requires any text to be put before the end of the file, denoted by 'exit 0'. Open the required file for editing:

```
sudo nano /etc/rc.local
```

For Rev 1. Raspberry Pi boards add the following text:

```
echo pcf2127a 0x51 > /sys/class/i2c-
adapter/i2c-0/new_device
( sleep 2; hwclock -s ) &
```

For Rev 2. Raspberry Pi boards add the following text:

```
echo pcf2127a 0x51 > /sys/class/i2c-
adapter/i2c-1/new_device
( sleep 2; hwclock -s ) &
```

Then reboot:

```
sudo reboot
```

Step 3 - using the RTC

After you reboot the Raspberry Pi you should be able to access the module using the hwclock command. The first time you use the clock you will need to set the time. To copy the system time into the clock module:

```
sudo hwclock –w
```

To read the time from the clock module:

```
sudo hwclock -r
```

To copy the time from the clock module to the system:

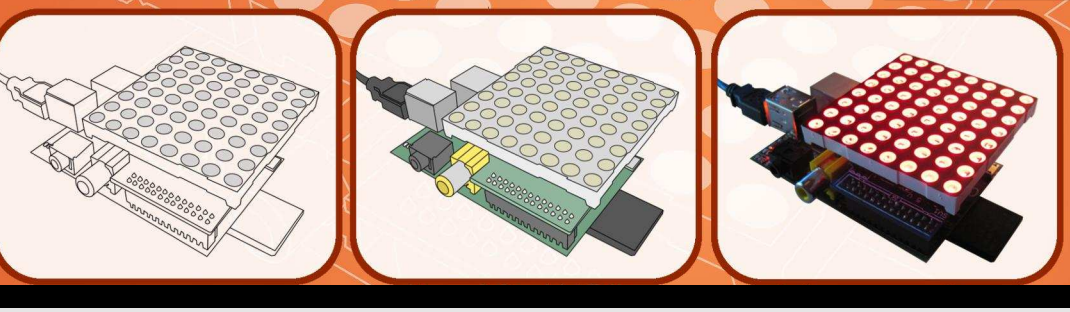
```
sudo hwclock –s
```

That's it... you can now keep time using your Raspberry Pi with no internet! Type hwclock into your resident search engine for a load more useful commands!

MODMYPI

This article is sponsored by ModMyPi

All breakout boards and accessories used in this tutorial are available for worldwide shipping from the ModMyPi webshop at www.modmypi.com



Bruce E. Hall
W8BH
Guest Writer

Part 3: Building a toolkit of patterns including Cylons!

DIFFICULTY : MODERATE

Introduction

In Part 1 and Part 2 of this series, we looked at how to build the Pi Matrix and how to program simple display routines in Python. In this tutorial we'll develop those routines further, giving us a whole toolkit of interesting displays to choose from.

Simplify, simplify

There are many ways to encapsulate data and code. I decided to just make some useful functions that anyone can grab and add to a project.

Let's start with some low-level routines that write data to the MCP27013 chip. In Part 2 we made a column-writing routine to turn on a single column, like this:

```
def SetColumn (col):
   bus.write_byte_data(ADDR,PORTB,0x00)
   bus.write_byte_data(ADDR,PORTA,0x80>>col)
```

All of the routines that write to the chip will look like this, writing byte values to one or more chip registers. They work fine as is, but I decided to simplify them further by taking out the repetitive parts, like this:

```
def Write (register, value):
   bus.write_byte_data(ADDR, register, value)

def SetColumn (col):
   Write (PORTB, 0x00)
   Write (PORTA, 0x80>>col)
```

We don't need to do this, but it moves all of the I²C calls to a single routine and removes the possibility of sending some of our data to the wrong I²C bus address. The simplest routine is one that writes data to the LED column and row inputs. Once we write that, we can further simplify SetColumn like this:

```
def WriteToLED (rowPins, colPins):
   Write (PORTA, colPins)
   Write (PORTB, rowPins)

def SetColumn (col):
   WriteToLED (0x00, 0x80>>col)
```

Again, this abstraction is not necessary but helpful. With WriteToLED we no longer have to remember whether columns go on PORTA or PORTB. Try it!

Patterns

We still have to remember that rows are active LOW and that the column bits are reversed. Wouldn't it be great to call a function with the first row bit and first column bit set and have the top left LED light up? It's easy! We'll use what we know and then never have to remember it again:

```
def SetPattern (rows, cols):
   WriteToLED(~rows, ReverseBits(cols))
```

The tilde '~' negates the row bits, making them all active LOW. We need to write a routine to reverse the order of the column bits, but that's all it takes. For example, suppose we want to make a small 2x3

rectangle. We need to set row bits 1 and 2 (0b00000110 = 0x06) and column bits 1, 2 and 3. (0b00001110 = 0x0E). Try calling SetPattern with some different bit patterns on the rows and columns to see what you can make, including SetPattern(0,0).

Orientation

The cables around my Raspberry Pi hold it in an upside-down orientation. By that, I mean the USB ports are on the left and the GPIO pins are facing me on the bottom. I was constantly turning my head around, trying to see if the correct LEDs were lit. Necessity is the mother of invention, so what would it take to flip the display?

Things get a little confusing here but when you flip a display, up means down and left means right! Something in the top-left, when flipped 180 degrees, is now bottom-right. We can do this by reversing the row and column bits:

```
def SetPattern180 (rows,cols):
   SetPattern(ReverseBits(rows), \
    ReverseBits(cols))
```

We can also change the orientation sideways left and right (90 and 270 degrees) by swapping rows and columns and reversing one of them. Since we are already reversing the columns in our original SetPattern function, we can combine both operations in a new SetPattern function, like this:

```
def SetPattern (rows,cols,orientation=0):
   if orientation==0:
      WriteToLED(~rows,ReverseBits(cols))
   elif orientation==90:
      WriteToLED(~cols,rows)
   elif orientation==180:
      WriteToLED(~ReverseBits(rows),cols)
   elif orientation==270:
      WriteToLED(~ReverseBits(cols), \
       ReverseBits(rows))
```

Our routine bulked up a bit, but now we can use the Raspberry Pi orientated at 0, 90, 180 and 270 degrees. Notice the default value of 0 in the definition; we don't even need to specify any orientation at all, if the Raspberry Pi happens to be right-side up.

Back to basics

Now that we can create any pattern in positive logic and in any orientation, it is much easier to set individual LEDs plus columns and rows of LEDs.

```
def SetLED (row,col):
   SetPattern(1<<row, 1<<col)

def SetColumn (row,col):
   SetPattern(0xFF, 1<<col)

def SetRow (row,col):
   SetPattern(1<<row, 0xFF)
```

On my upside-down board, I set the orientation in SetPattern to 180 instead of 0.

Play time!

In Part 2 we created some test patterns to exercise the Pi Matrix. Here are a few more. One of my favorites is the Cylon from the TV series Battlestar Galactica. This had a red eye that shifted position back and forth across its face.

```
def MultiRowCylon (pattern,numCycles):
   for count in range(0, numCycles):
      for col in range(0,7):
         SetPattern(pattern, 1<<col)
         time.sleep(delay)
      for col in range(7,0,-1):
         SetPattern(pattern, 1<<col)
         time.sleep(delay)

def SingleRowCylon (row,numCycles):
   #Side-to-side LED chaser, single row
   MultiRowCylon(1<<row, numCycles)

def AllRowCylon (numCycles):
   #Side-to-side LED chaser using all rows
   MultiRowCylon(0xFF, numCycles)
```

You can do some really fun animations just by experimenting and trying different patterns. Check out my demo video at `http://youtu.be/VbPBNmlGy34`.

The sample code for this month is too long to publish in the magazine, but you can download it from `http://w8bh.net/pi/matrix3.py`. (You may need to change the ORIENTATION constant at the start of the file. Also, Model B Revision 1 owners need to set `bus=smbus.SMBus(0)` near the end of the file).

Next time we will learn how to scroll text and make a marquee display. Have fun!

Getting MAME up and running

DIFFICULTY : INTERMEDIATE

Karl Welsh

Guest Writer

When Arcades Ruled

I remember the first video games arriving in the late 1970's. Historically, amusement arcades were located at holiday destinations and filled with electro/mechanical games and pinball machines dating back to the 1950's and 60's.

Arcades changed with the introduction of the first commercially successful 'discrete logic' or 'digital' video game, Pong (Atari Inc: 1972), and then with second generation CPU powered machines: Space Invaders (Taito/Midway: 1978), Asteroids (Atari Inc.: 1979), Galaxian (Namco/Midway: 1979), Defender (Williams Electronics: 1980), Pac-Man (Namco/Midway: 1980) and Donkey Kong (Nintendo: 1981).

A behemoth of an industry was born. Arcade games began appearing everywhere. Dedicated arcades in towns and cities, game cabinets in petrol stations, supermarkets, restaurants, pubs/bars. I used to frequent a chip shop just because it had Centipede (Atari Inc: 1980).

At its peak in 1981, arcade games generated annual revenues of over $5 billion in the USA ($12.79 billion in 2013 dollars). In 1983, the USA endured 'The Video Game Crash'. Financial markets lost faith in the 'passing fad' of video games. The 'Golden Age of Video Games' was over.

Retro gaming, emulation, and MAME

Peter De Vries, an American editor, novelist, and satirist wrote 'Nostalgia isn't what it used to be', but I disagree. 'Old School' or 'Retro Gaming' has a substantial following and not just because of nostalgia, but due to the restrictive/primitive nature of the hardware. Many 'Retro Games' are defined by their elegant simplicity and gameplay. Atari Inc. co-founder Nolan Bushnell's quote that the perfect game is 'Easy to Learn, Difficult to Master' describes the classics from the 'Golden Age of Video Games'.

An emulator is hardware or software, or both, that duplicates the functions of one computer system (the guest) in another computer system (the host), different from the first one. The emulated behavior closely resembles the behavior of the real system (the guest). When running an emulator such as MAME (Multiple Arcade Machine Emulator), it may seem odd that some games may be slower on the Raspberry Pi. Despite the increase in processing power, 32Bit ARM vs 8Bit Z80/6502 or 16Bit 68000, emulation is VERY processor intensive. A 'host'

system often has to be many times more powerful than its 'guest' system.

MAME was started in 1997 by the Italian programmer Nicola Salmoria, to emulate the hardware of arcade game systems. It was originally written for MS-DOS to emulate Pac-Man, Pengo (Sega: 1982), Crazy Climber (Nichibutsu: 1980), Lady Bug (Universal/Taito: 1981) and Rally X (Namco/Midway: 1980). It now supports over 10,000 ROMS (although not all are playable). With MAME you play the original game code of classic arcade games without the highly collectable and expensive original cabinets, or inserting any coins!

ROMS and Samples

ROM files will be required to run specific games. There are some ROMs available for non-commercial use at http://mamedev.org/roms/ MAME requires the correct ROM revision for individual versions of the core program. Luckily, it will display and name the missing ROM files, so just try another ROM revision.

Some early arcade games used additional 'Discrete Logic' circuits for sound (Astro Blaster (Sega: 1981), Berzerk (Stern Electronics: 1980), Donkey Kong (Certain Sounds e.g. The 'Jump'), Gorf (Bally Midway: 1981) etc). MAME cannot Emulate these so samples are required to run correctly.

Most games up to the mid/late 1980s are fine, but some require additional emulation processing due to additional 'Custom' processors in the original cabinet.

There is a list of some great games with the game name, MAME code, revision required and whether 'samples' are needed at:
http://www.raspberrypi.org/phpBB3/viewtopic.php?f=78&t=29427
To run many of these MAME ROMs, a license is required,
http://www.mamedev.org/devwiki/index.php?title=FAQ:ROMs

Compiling AdvMAME

AdvMAME can be compiled on a 256 or 512 MByte Raspberry Pi. These instructions have been tested with Raspbian Wheezy. Open a LXTerminal window from the menu. Then before continuing, make sure the Raspbian installation is up to date,

```
sudo apt-get update
sudo apt-get upgrade -y
```

Download AdvMAME version 0.106.0 (advancemame-0.106.0.tar.gz) from,
http://sourceforge.net/projects/advancemame/files/advancemame/0.106.0/
This older version will produce better results on the Raspberry Pi. Newer MAME releases offer greater accuracy in emulation, but not optimisation of performance.

Compiling AdvMAME is resource intensive. Therefore, before compilation use

```
sudo raspi-config
```

to disable X on boot. This will allow the compilation to use the physical memory on top of the ARM chip, rather than accessing swap space on the SD card. With no overclocking, it will take around six hours to compile. Some overclocking will improve the performance of the emulator. Select the highest turbo overclocking mode that is stable on your Raspberry Pi. Compilation is possible with 64MBytes of RAM allocated to the GPU (default setting) on a 256MByte Raspberry Pi. If more memory is allocated to the GPU, it will start to use the swap space a lot more and may fail to compile. After completing the configuration, exit raspi-config and reboot.

Login using your account and password. Install the dependencies libsdl1.2-dev and gcc-4.7

```
sudo apt-get install -y libsdl1.2-dev \
gcc-4.7
```

The gcc-4.7 compiler is needed to compile

AdvMAME with the least additional effort. (It is only possible to use the standard system compiler if several of the AdvMAME source files are modified.) To use the gcc 4.7 compiler type,

```
export CC=gcc-4.7
export GCC=g++-4.7
```

Then unpack the AdvMAME source code, configure it and compile:

```
tar xvfz advancemame-0.106.0.tar.gz
cd advancemame-0.106.0
./configure
make
```

The compilation will take four and a half to six hours, depending on your overclocking setting. When the compilation has finished type

```
sudo make install
```

This is optional, but is useful as it negates the need to change directory or set the PATH manually.

Configuring AdvMAME

AdvMAME should be run once to setup all the correct folders and configuration files. Type

```
advmame
```

It will give you a message telling you that it has set up the default options. Now type

```
startx
```

to start X. There should now be a hidden folder .advance in your home directory (/home/pi by default). If you cannot see it, right-click your mouse and tick the 'show hidden folders' box.

Now download some ROMS. As an example, the Star Fire ROM can be downloaded from http://mamedev.org/roms/starfire/starfire.zip

Then either use the file manager or type

```
mv starfire.zip ~/.advance/rom/
```

to move it into place. If the selected ROM has samples, put those in the samples/ directory. There is no need to uncompress them!

Next, open

```
.advance/advmame.rc
```

and add either

```
device_video_clock 5 - 50 / 15.62 / 50 ;
5 - 50 / 15.73 / 60
```

(on one line) for HDMI output or

```
device_video_clock 5 - 50 / 15.73 / 60
```

for composite output. Then change the lines

```
display_resize mixed
display_artwork_backdrop yes
display_artwork_overlay yes
```

to

```
display_resize fractional
display_artwork_backdrop no
display_artwork_overlay no
```

Then save and close advmame.rc. MAME can be run from X, but using the console will significantly increase the performance and provide a fullscreen display. Quit X.

Running MAME

Now that MAME has been fully configured, type

```
advmame Name_of_ROM
```

where Name_of_ROM is the name of the ROM rather than the game title. For example,

```
advmame starfire
```

To play other games, download them and put the

ROMs into the .advanced/rom/ directory.

Controls

Control is with the cursor keys. The fire and jump buttons are generally **Left-Ctrl**, **Space** and **Left-Alt** respectively. Joysticks and gamepads can also be configured and used. All options for control settings, video, sound, dip switches etc. are easily modified in the options menu and can be saved for general 'ALL Games' or individual games 'This Game'. Other controls are:
5 - Add Coins
1 - 1 Player
2 - 2 Players
TAB - Options Menu (use Cursor Keys and Enter)
ESCape - Exit

Other configuration options

MAME settings can be infinitely customised. The defaults in advmame.rc are perfectly acceptable for several games, but there are a few exceptions:

1. The default 'display_color' is 'auto'. Vector games - Asteroids, Star Wars, etc. will suffer from an incorrect application error, so DON'T change advmame.rc! In the Options Menu (TAB) navigate to Video, Color and change 'auto' to 'bgr16' then 'Save for this Game'.

2. The default 'display_mode' is 'auto' and 'display_magnify' is '1'. Certain games suffer from an incorrect application error in Aspect Ratio (e.g. Burger Time (Data East: 1982), I-Robot (Atari Inc: 1983) and Track & Field (Konami: 1983)). Again, DON'T change advmame.rc! In the Options Menu (TAB) navigate to Video, magnify and change '1' to '2' then 'Save for this Game'

3. Consider changing 'display_resizeeffect' from 'auto' to 'none'. Personally, I'm not a fan of any of these in emulations and they will be applied when changing the magnification mode as described above. However, they can be scrolled through in the video options menu. Experiment, and see if you like any of them!

4. If you are old enough to remember amusement arcades, did the SAME game seem harder in different venues? Well, it probably WAS! Inside the cabinet are toggle 'DIP' switches that could be set to change difficulty, lives, extras, etc. These are emulated in MAME and can be configured - options (TAB), Dip Switches.

Don't Like the Command Line?

The Advance Suite of software also includes a fully integrated front end for AdvMAME, AdvMENU,
http://sourceforge.net/projects/advancemame/files/advancemenu/
A front end is a G.U.I (Graphical User Interface), which elevates the troublesome command line typing for emulator control and execution. The front end can also be helpful when your ROM collection gets larger, since it can become hard to remember all the MAME codes (or what ROMS are in the folder!). This can be installed using the exact same method as MAME itself. Then just run the executable advmenu.

Older versions, better performance

Older versions of AdvMAME can often give improved results, but require changes to the scripts to compile correctly on the Raspberr Pi. I suggest downloading 0.94.0, as it requires only a few script changes. First, modify /advance/linux/vfb.c. Change 'MAP_SHARED ! MAP_FIXED' to 'MAP_SHARED,' save and close and compile as above.

In 1999, Billy Mitchell achieved the first perfect PacMan score of 3,333,360 by eating every possible dot, power pellet, fruit, and enemy!

Steve Wiebe beat Billy's 1982 Donkey Kong record. Many others have tried to get higher. Hank S Chien is currenly the champion with. 1,138,600 points.

XML for the Raspberry Pi: Part 1

DIFFICULTY : INTERMEDIATE

John Shovic

Guest Writer

Introduction

This series of articles will discuss the use of XML on applications for the Raspberry Pi. Part One covers what is XML and the format of the data structures. Part Two will cover building and parsing XML in Python and Part Three will show how XML is used as a communications protocol for a client / server application, RasPiConnect. RasPiConnect is an iPad/iPhone app that connects and displays information for any number of Raspberry Pi's via a defined XML interface.

What is XML?

XML stands for eXtensible Markup Language. It is a language to structure, store and transport information in a hardware and software independent way. It kind of looks like HTML but it is used to transport information not to display information. HTML and XML are both examples of an SGML (Standard Generalized Markup Language).

What do you use XML for?

It is a little difficult to understand, but XML does not "Do" anything. XML is designed to transport information unlike HTML which is used to display information. You use XML to structure data (usually in a human readable format) and to send this data to other pieces of software on your own machine or across the Internet. Often user preferences or user data is also stored in XML and then written to files. If you need to send structured data, then XML is an excellent choice. It is easy to parse, easy to modify, and most importantly, easy to debug. One very useful characteristic of XML files is that they can be extended (more elements, attributes, etc.) without breaking applications. Providing, of course, those applications are well written (see Part Two of this series).

Here is a complete XML message:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLCOMMAND>
    <OBJECTID>12</OBJECTID>
    <OBJECTSERVERID>BL-1</OBJECTSERVERID>
    <OBJECTTYPE>2048</OBJECTTYPE>
    <OBJECTFLAGS>0</OBJECTFLAGS>
    <RASPICONNECTSERVERVERSIONNUMBER>2.4
    </RASPICONNECTSERVERVERSIONNUMBER>
    <RESPONSE>
        <![CDATA[100.00, 0.00, CPU Load]]>
    </RESPONSE>
</XMLCOMMAND>
```

Structure of an XML message

Unlike HTML, in XML you define your own tags. A well formed XML message has a "root" and then "branches" and "leaves".

The first line is the XML declaration. It rarely changes. The second line describes the root element of the XML document.

```
<XMLCOMMAND>
```

Note that the end of the XML root has a closing tag:

```
</XMLCOMMAND>
```

All elements in XML must have an opening and closing tag. This, in addition to the root is the definition of a "well-formed XML document". By the way, all tags in XML are case sensitive. A good XML coding practice is to make all of the tags uppercase. Doing this also makes the structure of the XML stand out when you read it.

Add child elements

Child elements are used to provide additional data and information about the enclosing XML element (i.e. <XMLCOMMAND> in the example above). Note that XML does not require the same set of child elements for each enclosing XML element, making upgrading or changing your elements easy. However, your parser does have to handle this situation!

Child elements are XML elements underneath the root (`OBJECTID`, `OBJECTSERVERID`, `OBJECTTYPE`, `OBJECTFLAGS`, `RASPICONNECTSERVERVERSIONNUMBER`, `RESPONSE`). All of these tags must have a beginning and ending tag similar to the root. In addition, all elements can have child elements nested inside.

XML attributes

XML elements can have attributes, just like HTML. Attributes provide additional information about an element. By convention, attributes are usually given in lower case.

```
<PICTURE id="1" type="gif" file="BPNSCFA.gif">
</PICTURE>
```

It is good practice to use attributes in XML sparingly and in a consistent manner. You can rewrite the above XML as the following:

```
<PICTURE id="1">
    <TYPE>gif</TYPE>
    <FILE>BPNSCFA.gif</FILE>
</PICTURE>
```

Not having attributes makes the parsing of the XML easier in many ways.

XML special character secrets

There are two characters that are not allowed inside of an XML element. They are the "<" and "&". The ">" character is allowed, but it is also good practice to replace this character. The pre-defined entity references in XML for these characters are "<", "&" and ">".

Sending special data in XML

Sometimes you want to send general data in your XML element without replacing special characters. For example, you might want to send an HTML page inside an XML element (the RasPiConnect application does this) and you don't want to change all the characters. XML parses all text inside elements by default, but there is a way to change that: CDATA. Inside a CDATA structure, the XML parser ignores the data and it can be passed without change in an XML message. CDATA looks like this:

```
<![CDATA[<XML & DOES & NOT <LIKETHIS>]]>
```

Validate your XML

There are many sites on the web that will validate that your XML is well formed. http://www.xmlvalidation.com is one such site. Cut and paste the XML from the first page to try it out.

Coming in Part Two and Three

Part Two of this series will describe simple ways of generating and parsing XML in Python on the Raspberry Pi. Part Three will look at an actual application and how the same XML is used on both the iPad and directly on the Raspberry Pi.

Conclusion

XML is a simple, easily understood method for sending information in a hardware and software independent manner. The main advantages of XML are readability and portability between systems. It provides an easily extensible framework for information interchange.

To learn more about XML try the following websites:

http://www.w3schools.com/xml/
http://www.quackit.com/xml/tutorial/

RasPiConnect

Connect your Raspberry Pi to the World!

Available on the App Store

RasPiConnect connects your Raspberry Pi to the outside world. It allows you to control virtually anything you connect to your Raspberry Pi from your iPad or iPhone.

- EASY to setup - no syncing required
- Buttons, gauges, webpages, webcam pictures and more!
- Build your pages on your iPad/iPhone
- Supports multiple Raspberry Pis
- Five pages of control panels
- Unlimited Controls
- Exchange your panels with friends
- Supports any computer that supports Python (windows, linux, etc.)

MiloCreek
Where Innovation Flows

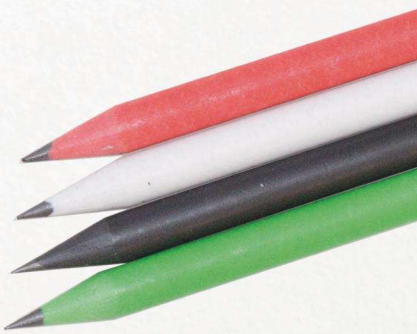
The Raspberry Pi is helping millions of kids write their first

print 'hello world'

We'd like to thank you for
making cool projects,
spreading the word,

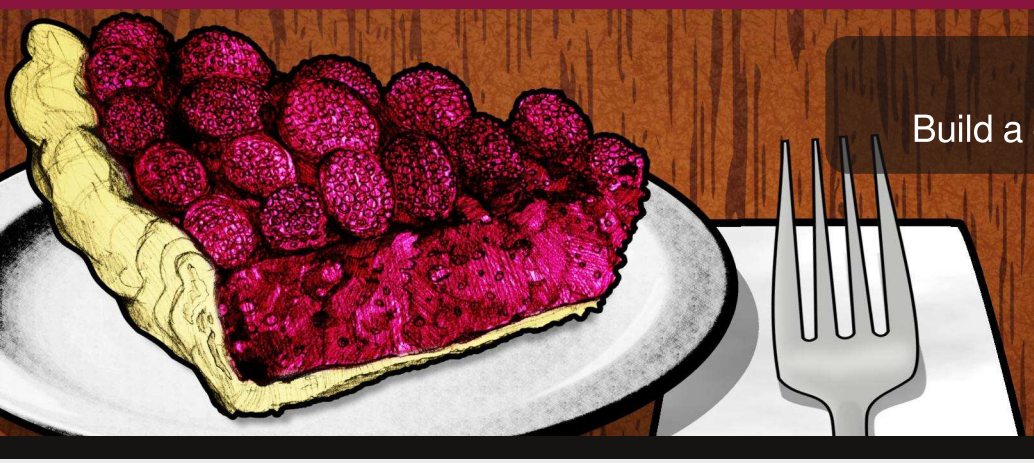
...and also for buying sweet, sweet swag

Your generous support helps us do more*
http://swag.raspberrypi.org



AUTHENTIC SWAG EDUCATING SINCE '12

*Hello World is alright, but we've got to teach kids "20 GOTO 10" as well



Bake your own Pi filling

DIFFICULTY : INTERMEDIATE

Martin Kalitis

Guest Writer

By now you've probably tried at least one version of an operating system on your Raspberry Pi. If you read issue 12's article on operating systems, you will realise there are many variations in existence.

What if you needed some customizations that are not available in the various images out there? Perhaps you want to build your very own version for a custom purpose such as a DHCP server. Or maybe you just want a greater understanding of what goes into creating one of the many software images out on the web.

This is the first in a series of articles where we start with raw ingredients, baking them to perfection, producing customised images for your Raspberry Pi. Along the way we will explore the multitude of options available and what processes are happening under the hood.

The per-requisites for this article are as follows:

1) A PC capable of running Linux
2) Ubuntu 12.10 or 13.4 base installation and an internet connection.
3) A SD card reader
4) A SD Card (2GB should be fine, larger is good too:))

Building Linux and all of the other components from source code is a big job and can take many hours; on a small computer like the Raspberry Pi this could turn into days. To reduce the amount of time this takes we will be using a desktop PC running Ubuntu Linux for the job. This machine has an Intel CPU instead of an ARM CPU and the compilers that it uses will only produce software that will run on an Intel system.

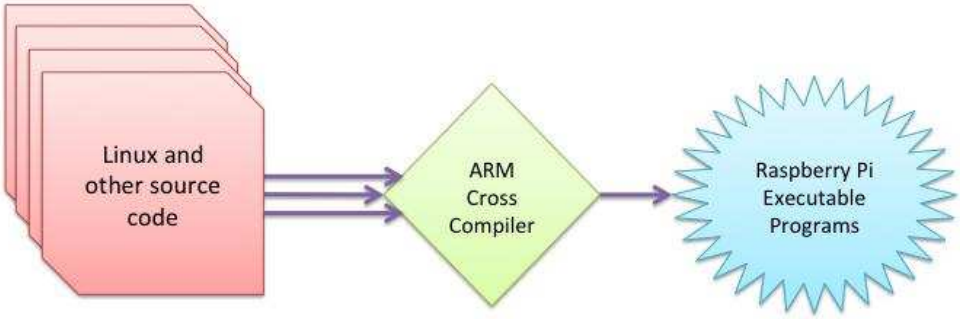
Creating the ARM cross compiler

To overcome this problem we need to use a compiler that is capable of creating software that runs on the ARM CPU. This compiler is known as a cross-compiler and is capable of creating executable code for a CPU architecture that is different to the one the compiler is running on.

There are a number of pre-built options available, some free and others that attract a fee. As we are interested in doing as much as we can from the source code we will be building our own.

We now have to compile. It is important to ensure that not only do we have all the components needed for running Linux but that they are also compiled in the correct order and any fixes to each source package are applied where appropriate. To make this easier there are a number of tools available, both open source and commercial, that perform this function. These tools ensure that after the hours of building the software the result will run on the Pi. We will be using the open source Buildroot tool for compiling our custom images in this series of articles.

The ARM cross compiler building source code for the Raspberry Pi



Before we can begin with creating our images, we need to install a distribution on Linux on our Desktop PC and install the compilers and other tools that are used during the build process to make our images.

We start with a system that already has Ubuntu Desktop installed. It does not matter if it is a 32-bit or 64-bit installation however it must be version 12.04.2, 12.10 or 13.04. If we are installing a fresh copy of Ubuntu onto a PC then version 13.04 is recommended. All the commands that we need to configure our desktop system will be done from a terminal window so open up a terminal window. Let's get started.

It's always a good idea to make sure we have a system that is up-to-date prior to making any of these changes. This can be accomplished with the following command:

```
sudo apt-get upgrade -y
```

Next we install the compilers and other associated tools. These will be used for the compilation of the tool chain used to create the executable binaries that the Raspberry Pi can run.

```
sudo apt-get install build-essential
```

Next we need to install a few of support libraries and components that will be used by the cross compilation tool.

```
sudo apt-get install \
libncurses5-dev bison \
flex texinfo gawk
```

Finally we need components that provide the functionality to get the source code used to create our final operating system.

```
sudo apt-get install \
git mercurial subversion
```

In the next issue we will be using our newly configured Ubuntu system to compile and install a basic Linux system onto a SD card that we can use on our Raspberry Pi.

DID YOU KNOW?

The University of Cambridge Computer Lab offers a free cource on building a very simple operating system from scratch. They start by introducing assembly, http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/

Passing values and Add - part 2

DIFFICULTY : ADVANCED

Bruce Smith

Guest Writer

Learn how to program your favourite computer in its native machine code by using Assembly Language. This is the second article in the series. The first article was published in Issue 11 of the MagPi.

The ARM has a very specific and special design; this is known as its architecture because it refers to how it is constructed and how it looks from the user's point of view. Having an understanding of this architecture is an important aspect of learning to program Raspberry Pi's processor. You need to appreciate how it all fits together and how the various elements interact. In fact, the purpose of much of the machine code we will be creating is to gain access and manipulate the various parts in the ARM chip itself.

The ARM uses a load and store architecture. It is very efficient at manipulating data, doing so quickly. It does this by using special areas of memory built inside the processor called registers. Because they are on-board they are lightening fast in operation. For the most part, when we program in assembler we are doing so in User Mode – and as the sentence suggests there are other modes available. These are more specialised and they require a degree of programming experience to use them.

In User Mode there are 17 registers that can be accessed by the programmer, as shown in Figure 1. In their simplest form they are numbered 0 to 15 and we prefix them with an 'R' to indicate a register is being referred to - thus R0, R1, R2. Registers R0 to R12 are available for all your requirements and, for the most part, can be treated as identical. You can load, manipulate and store information in these. Generally these registers are used to hold an item of data or a value that represents a location in memory (an address).

Registers R13, R14 and R15 all have special functions. R13 is the Stack Pointer and this contains an address that points to an area of memory which we can use to save information. (This area of memory is called a stack and it has some special properties we will look at in a future article.) We saw R14 and R15 in action in the last article (and in the program examples in this one) with the instruction:

```
MOV R15, R14
```

It's generally the very last one that you will use in

your assembler programs. When a machine code program is called from the command line, the address calling the program is placed in R14 – the Link Register. (Remember that although you are using BBC BASIC to assemble your program it is itself being run by machine code, so we need to know where to return to when your code has finished executing!) The MOV instruction moves this saved address into R15 – the Program Counter (PC). The PC is the means by which the ARM knows where it is in a seemingly never ending list of machine code instructions. If you corrupt the contents of the PC then your program will certainly 'freeze', so good housekeeping in this respect is imperative.

The 17th register in User Mode is called the Current Program Status Register or simply, Status Register (SR) and this register allows you as the programmer to test and control your program operation – we'll look at this in detail next time.

Each of these registers can hold a value that equates to 32-bits or four bytes in length, also known as a word. If you are not familiar with bits, bytes, words and binary numbers in general then check out the following link:
www.brucesmith.info/numbers.html

Passing Values in Registers

The registers R0 through R12 are identical for all intents and purposes. Human nature being what it is we tend to use the lower number registers more frequently. The other advantage of this is that we can pass values from our BBC BASIC programs directly into registers.

When CALL is used to execute a program the values of the integer variables A% through to H% are passed into registers R0 to R7 respectively. The program VARIABLETEST will add the values passed to it by B% and C% and the result is printed out using the OS_WriteC call. The result returned is the same as last month's sample program.

The CALL command cannot return a value to your calling program, to do this you should use the USR command. This is similar to CALL, however it allows you to pass a result back to BBC BASIC from your machine code. It uses the format:

```
<variable>=USR <address>
```

As you can see the address, which may be a variable, is used to identify where the machine code is located. A variable is specified on the left of the command, into which a value will be returned. For example:

```
Result=USR START
```

The value returned in the 'Result' variable is that held in R0 when the code hands control back to BBC BASIC.
The program USRTEST modifies VARIBLETEST slightly to add the values passed in B% and C% storing the result in R0 so that it is captured in the Return variable.

BREAK OUT PANEL 1:
Programs and Description

```
10 REM >VARIABLETEST
20 DIM CODE% (100)
30 B%=&20
40 C%=&A
50 P%=CODE%
60 [
70 .START
80 ADD R0, R1, R2
90 SWI "OS_WriteC"
100 MOV R15, R14
110 ]
120 CALL START
```

When the program is called B% and C% are placed in R1 and R2 (lines 30 and 40). The ADD instruction in line 80 places the sum of R1 and R2 into R0. So &20 is added to &A to give &2A,

Line 90 uses the OS Write Character call to print the contents of R0 as an ASCII character. The ASCII character for &2A is '*'.

```
10 REM >USRTEST
20 DIM CODE% (100)
30 B%=&20
40 C%=&A
50 P%=CODE%
60 [
70 .START
80 ADD R0, R1, R2
100 MOV R15, R14
110 ]
120 Result=USR START
130 PRINT Result
```

Line 90 has been deleted from the original program. Line 120 has been modified to replace CALL with USR and line 130 has been added to print out the value of 'Result'. The value returned is 42 which is the decimal equivalent of &2A.

BREAK OUT PANEL 2:
The ADD instruction

The ADD instruction is a very commonly used one in ARM machine code. It takes the format:
ADD <Destination>, <Operand1>, <Operand2>
<Destination> is the destination register for the result of the sum of <Operand1> and <Operand2>, If you like:
<Destination>=<Operand1>+<Operand2>.
The <Destination> must always be a register but <Operand1> and <Operand2> can be registers or immediate values which are signified with a hash. <Operand1> may be the same as <Destination>. Here are some examples:

```
ADD R0, R0, #1 ; Increment contents of R0 by 1
ADD R0, PC, #12; Add 12 to value of PC and save
in R0
```

TIP PANEL

The BBC BASIC Assembler will recognise both lower and uppercase use of 'R' when you reference them in your assembler. So R0 and r0 are interchangeable. The assembler will also allow you to use 'PC' or 'pc' in place of R15, and likewise 'LR' or 'lr' and 'SP" or 'sp" for R13 and R14 – these being the link register and stack pointer. The assembler will assemble these following instructions identically:

```
MOV R15, R14
MOV pc, lr
```

From a programming style point of view it is best to stick with one or the other. The advantage of this is that you do not continually have to seek out the SHIFT or CAPS LOCK key. If you are a proficient typist them you will find a combination of upper and lower case names, say lower case for labels, is more pleasing to the eye.

R0	Available
R1	Available
R2	Available
R3	Available
R4	Available
R5	Available
R6	Available
R7	Available
R8	Available
R9	Available
R10	Available
R11	Available
R12	Available
R13	Stack pointer
R14	Link register
R15	Program counter
Current program status register	

Figure 1. Registers available to the programmer in User Mode.

Bruce Smith is an award winning author. His book, Raspberry Pi Assembly Language Beginners, is now available from Amazon. Check him and his book out at www.brucesmith.info for more Raspberry Pi resources.

The MagPi What's On Guide

Want to keep up to date with all things Raspberry Pi in your area?

Then this section of The MagPi is for you! We aim to list Raspberry Jam events in your area, providing you with a Raspberry Pi calendar for the month ahead.

Are you in charge of running a Raspberry Pi event? Want to publicise it?
Email us at: editor@themagpi.com

Preston Raspberry Jam #RJam

When: Monday 5th August 2013 @ 7.00pm
Where: Media Innovation Studio, Media Factory, Preston, PR1 2HE

This event is free and will include a number of short talks and demonstrations in relation to the Raspberry Pi. Further information and tickets available at http://raspberryjam10.eventbrite.com

Manchester Raspberry Jam XIV

When: Saturday 17th August 2013 @ 10.00am
Where: Madlab, 36-40 Edge Street, Manchester, M4 1HN

The event will run from 10am until 5pm. Tickets available to purchase at:
http://mcrraspjam.org.uk/next-event

TechnoMach Summer Event 2013

When: Saturday 24th August @ 11.00am
Where: Ysgol Bro Ddyfi, Greenfields, Machynlleth, Powys, SY20 8DR

Runs from 11am until 3pm. Free entry but donations welcome. Further information:
http://technomach-eorg.eventbrite.com

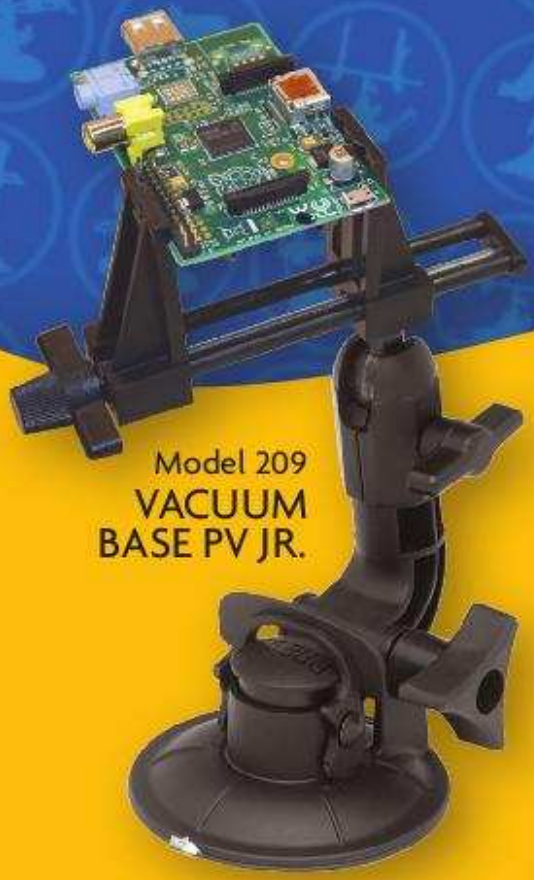
e-day2

When: Saturday 7th September 2013 @ 10.00am
Where: Gateshead Central Library, Prince Consort Road, Gateshead, NE8 4LN

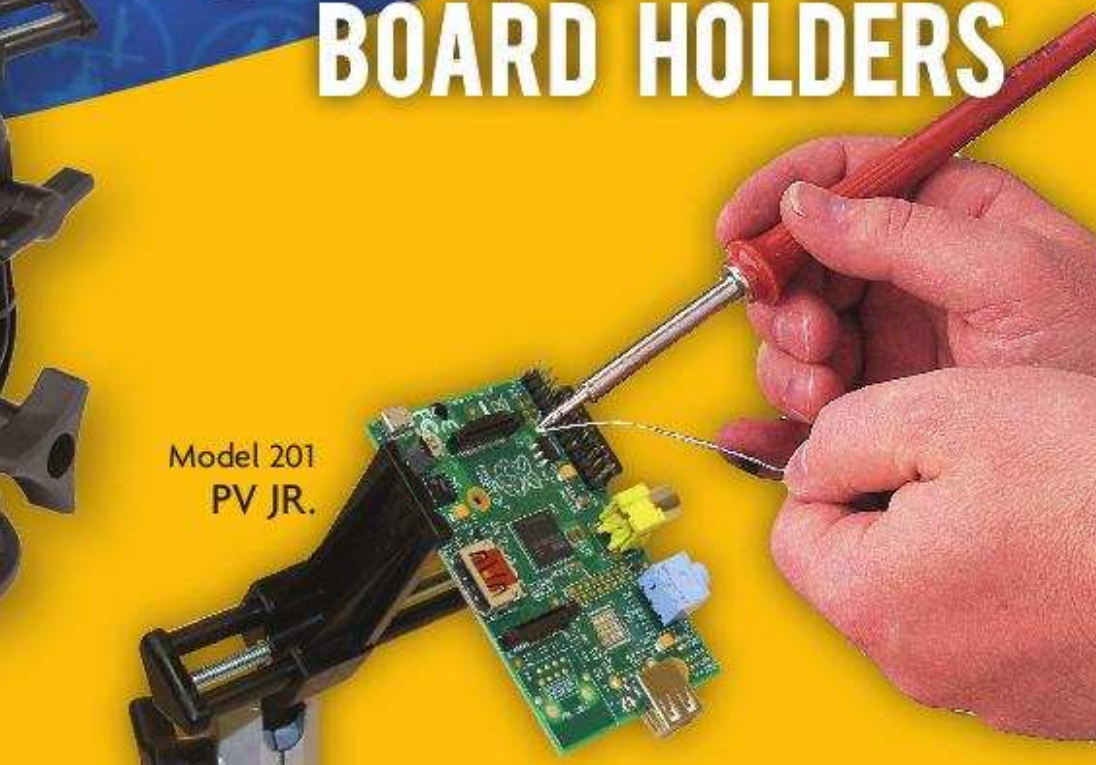
The event will run from 10am until 3pm. Further information and free tickets are available at:
http://www.eventbrite.co.uk/event/5720145108

VISIT US ON

WORLD'S MOST VERSATILE CIRCUIT BOARD HOLDERS

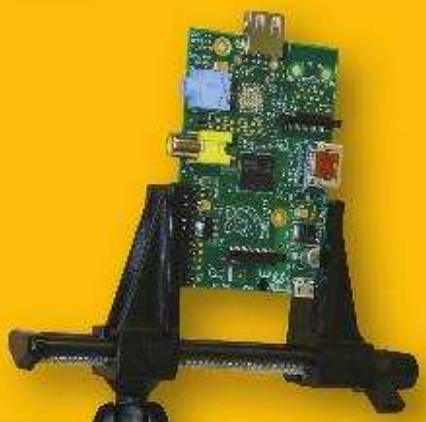


Model 209
VACUUM
BASE PV JR.



Model 201
PV JR.

- Work-holding tools for electronics projects
- Circuit board holders make soldering easy & fun
- Versatile hobby vises for any project



Model 207
VISE BUDDY JR.

PANAVISE®
Innovative Holding Solutions

www.panavise.com
00 1 800 759 7535
7540 Colbert Drive • Reno • Nevada • 89511 • USA

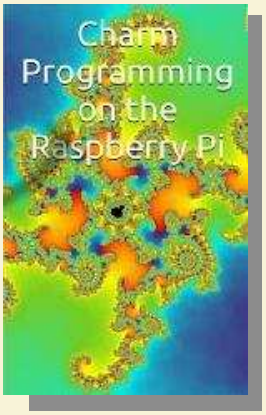
Charm Programming on the Raspberry Pi
SPeter Nowosad
Kindle Edition

This book is a nice companion to the Charm articles published in The MagPi.

The book begins with an excellent explanation of the differences between assembly and programming languages, and how they intereact. This is followed by the history of Charm, which was developed by the author of this book, running the RISC OS operating system. It is a charming story that will encourage others to play around with programming languages and possibly develop something new. The rest of the book is a reference guide to Charm with explanations of each aspect of the language and the obligatory "Hello world" program. The book finishes with Charm demo programs, such as (Chinese) checkers and hanoi. The book comes with an accompanying website (http://charm.qu-bit.co.uk/), which contains a download repository and the ability to contact the author with any queries or suggestions.

While some of the English language using could be quite daunting to programming novices, this book contains excellent computer theory, which really explains the concept of computer programming to the beginner, but is still vigorous enough for the expert.

If you are interested in learning more about Charm this is a good reference book to add to your Raspberry Pi/programming library!



POWERING THE PI FOR PORTABLE PROJECTS

RS Components has unveiled three rechargeable battery packs to keep you powered up while you're programming.

These new power banks are designed to meet the different requirements of the user. They range from the low-cost lightweight 2200mAh (min) Lithium-Ion version to the larger higher capacity 5000mAh (min) Li-polymer and 10400mAh (min) Lithium-Ion models with extended cell life for more power-hungry applications.

Each model is equipped with dual USB ports which means you can power your Raspberry Pi while the power bank itself is charging. A micro USB cable is standard with the battery pack.

The power bank automatically detects the output current of the connected power supply before charging. Safety of the user is a key consideration with over charge, over discharge, over current and short circuit protection.

These power banks can be used with most digital devices with DC 5V input - such as tablets, smart phones, portable MP3 players and GPS.

The three power bank models PB-2200, PB-5200 and PB-H10400 are all available to purchase direct from RS stock.

GET NEXT ITEM

Introducing __iter__(self)

Python iterators & generators

DIFFICULTY : BEGINNER

Alan Holt

Guest Writer

In this article we introduce Python iterators and generators. In computer science, an iterator is a container comprising a number of items. The items themselves are arbitrary and can be any data structure the language supports (they can even be other iterator objects).

Iterators can be processed using common methods regardless of the nature of the items within the iterator. In contrast, with programming languages like C, lists (arrays) are traversed by looping through an index sequence counter or by incrementing a pointer.

The C programmer has to know how many items there are (or test for a null pointer) in order to prevent looping beyond the end of the list. In contrast, iterators provide a way of signalling that the container has been exhausted, forcing the loop to terminate.

A generator is an abstract control mechanism that produces a sequence, returning a new value each time it is called. Generators in Python resemble regular functions. What distinguishes regular functions from generators is that generators relinquish control by calling yield instead of return. This allows generators to be resumed later on. Generators are typically used to implement iterators but they can be used to write simple coroutines for co-operative multitasking.

Examples

In programming languages like C and Pascal, the for statement iterates over a progression of numbers. The for-loop construct has a start value and a termination condition (plus an optional step value). Python's for statement is different, in that it iterates over a sequence (list, tuple or data dictionary). To illustrate, run the interactive Python interpreter and define a list:

```
>>> a = [1,2,3,4,5]
```

Now loop through the list (type CTRL-D after the "..."):

```
>>> for i in a:
... print i,
... ^D
1 2 3 4 5
```

Note, the trailing comma after the print statement line ensures that each item is printed on the same line (as opposed to separate lines).

The for statement generates an iterator object from the list by applying the iter() function. The

resultant object has a next() method which accesses items in the sequence one at a time. A StopIteration exception is raised when there are no more items. This, in turn, terminates the for-loop. To illustrate, we can create an iterator:

```
>>> x = iter('12345')
```

The first call to the next() method returns the first digit for the binary sequence:

```
>>> x.next()
'1'
```

Subsequent calls return the rest of the sequence:

```
>>> x.next()
'2'
>>> x.next()
'3'
>>> x.next()
'4'
>>> x.next()
'5'
```

A StopIteration exception is raised when we attempt to read beyond the end of the iterator:

```
>>> x.next()
Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
StopIteration
```

We can loop over the iterator with the for statement:

```
>>> for i in iter('12345'):
...   print i,
... ^D
1 2 3 4 5
```

Many Python objects have built-in iterators. For example, iterators, are evident in file handling methods. Create some arbitrary lines of text as data to write to a file:

```
>>> line = "Python iterators and generators"
>>> lines = "\n".join(line.split())
```

```
>>> lines
'Python\niterators\nand\ngenerators'
```

Now open a file, write the lines of text and close it:

```
>>> fd = open("title.txt", "w")
>>> fd.write(lines)
>>> fd.close()
```

Open the file again, this time for reading:

```
>>> fd = open("title.txt", "r")
```

Read each line of the file:

```
>>> for line in fd.readlines():
... print line.strip(),
... ^D
Python iterators and generators
```

Here the readlines() method returns a list (of lines), then the for statement loops through each line. This could present a problem for a large file, as the entire contents would be resident in memory. Go back to the start of the file and call the next() method a couple of times:

```
>>> fd.seek(0)   # go back to start of file
>>> fd.next().strip()
'Python'
>>> fd.next().strip()
'iterators'
```

This shows that a value (in this case a line of text) is produced each time next() is called. We can read the remaining lines with a for-loop:

```
>>> for line in fd:
...          print line.strip() ,
... ^D
and generators
```

A StopIteration exception terminates the loop. We do not see it because the exception is caught by the for statement. We can verify the exception is raised by calling the next() method to read beyond the end of the file:

```
>>> fd.next().strip()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
StopIteration
>>>
```

For a more graceful exit, use the try/except statement:

```
>>> try: fd.next().strip()
... except StopIteration: print "end of file"
...
end of file
```

Writing iterators and generators

In this section we show how to write iterators and generators. We write a function to compute factorial to illustrate. The factorial of a number is the product of all positive integers up to and including the number. Thus, 4 factorial is 1 x 2 x 3 x 4 = 24. Specifically, for this example, we are interested in generating a sequence of factorials from 0 to n.

Define the factorial function (either type it in at the Python command-line or save it to a file and import it):

```
def factorial (n) :
  fact = [1]
  count = 1
  while True:
    if count > n:
      return fact
    else:
      fact.append(fact[-1]*count)
      count += 1
```

Note that this is not an iterator, it is just a regular function that produces a list of the factorials (up to and including n). Generate the first five factorials (0 to 4):

```
>>> factorial(4)
[1, 1, 2, 6, 24]
```

The factorial function produces a sequence (a list). We can then loop through this list using the for statement:

```
>>> for i in factorial(4):
...   print i,
... ^D
1 1 2 6 24
```

The complete list is generated on execution of the for-loop. For long lists, this is an inefficient use of memory. Instead, it would be better to generate each item of the list upon each iteration of the for-loop. In short, we need an iterator rather than a function that merely returns a sequence. We could create an iterator with the iter() function, but this achieves little, as the entire sequence of factorials would be created in memory when we ran factorial(). Furthermore, the for statement runs iter() on the sequence anyway.

Implemented as an iterator, the factorial operation is defined as the class below:

```
class Factorial:
  def __init__(self,n):
    self.n = n
    self.fact = 0
    self.count = 0
  def next(self):
    if self.count > self.n:
      raise StopIteration
    self.fact *= self.count
    if self.fact < 1:
      self.fact = 1
    self.count+=1
    return self.fact
```

Declare an instance of the class:

```
>>> f2 = Factorial (4)
```

Now call the next() method a couple of times:

```
>>> f2.next()
1
>>> f2.next()
2
>>> f2.next()
```

```
6
```

However, when we attempt to loop through it (using a list comprehension construct), an exception is raised:

```
>>> f2 = Factorial(4)
>>> [i for i in f2]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: iteration over non-sequence
```

Our class may have a next() method but it also needs an __iter__() method to make it an iterator. We simply need to add the __iter__() method to the class:

```
class Factorial:
  def __init__(self,n):
    self.n = int(n)
    self.fact = int(0)
    self.count = int(0)
  def next(self):
    if self.count > self.n:
      raise StopIteration
    self.fact *= self.count
    if self.fact < 1:
      self.fact = 1
    self.count+=1
    return self.fact
  def __iter__(self):
    return self
```

Declare a new instance of the (modified) class:

```
>>> f3 = Factorial(4)
```

Now we can iterate over it, in this example, using a list comprehension:

```
>>> [i for i in f3]
[1, 1, 2, 6, 24]
```

When a regular subroutine returns, its stack frame is popped, consequently; the subroutine cannot be re-entered. Generator functions call yield() instead of return. The stack frame is retained, enabling the generator to be resumed. Generators are resumed by calling next(), whereupon they resume from the point just after the yield statement. Generators then continue to execute until the next yield() statement is called. To illustrate, the factorial() function above is re-written as a generator:

```
def factorial(n):
  fact = 1
  count = 0
  while True:
    if count > n:
      raise StopIteration
    else:
      if count != 0:
        fact *= count
      count += 1
      yield(fact)
```

Declare an instance of the generator and iterate over it:

```
>>> f4 = factorial(4)
>>> [i for i in f4]
[1, 1, 2, 6, 24]
```

Summary

Iterators in Python provide a "get next item" method (i.e. next()) as well as a means of signalling the end of the sequence (StopIteration exception). The inclusion of an __iter__() method in a class provides an interface to looping statements.

Iterative procedures can be developed without any knowledge of internal structure of the iteration object. Iterators are ideal for processing long (or even infinite) sequences because each item of the sequence is generated only when it is referenced. The entire sequence, therefore, does not have to be resident in memory.

Generators are functions that can be resumed. The yield() statement enables a function to be resumed each time the next() method is called. Generators are, therefore, more like coroutines than subroutines, and can be used for co-operative multitasking.

Feedback & Question Time

Great mag, reminds me why I was drawn to Linux in the first place now that other mags are all fading.

Langer, Steve G

Wrote my first Python game tonight, courtesy of @TheMagP1. By which I mean, I copied the code out of the magazine.

Simon Jones (@Tarnimus)

I will make a donation this afternoon.

Keep up the great work.

Love your magazine!

Jan van Kessel

Thank you very much for your amazing work! It reminds me when I was using the ZX81 way back and reading the magazines we had in France then...

I've already connected the RaspberryPi to our TV set and my 8 years old son is having a blast creating Scratch sprites and having them move around and change colors when they hit each other.

I'm trying to teach him new tricks once in a while and I'm having some fun too!

Jean-Christophe Helary

Comments on Kickstarter

Magpi Magazines - First 12 issues and binder finally arrived!

Excellent!!

Thank you very much.

Best wishes,
Alan Hunt

Got my magazines today! YEAH!

Nicole Qc

Just received my magazine bundle...looking forward to trying out some projects!

Whizzkids

Comments on Issue 14

Excellent can't wait to read it.

solar3000

The most interesting thing – the stonework in the background of the rather splendid picture of the author (ladies, form an orderly queue), is the southern pier of Ironbridge, nr Telford, the first, er, Iron Bridge.

James H

The MagPi is a trademark of The MagPi Ltd. Raspberry Pi is a trademark of the Raspberry Pi Foundation. The MagPi magazine is collaboratively produced by an independent group of Raspberry Pi owners, and is not affiliated in any way with the Raspberry Pi Foundation. It is prohibited to commercially produce this magazine without authorization from The MagPi Ltd. Printing for non commercial purposes is agreeable under the Creative Commons license below. The MagPi does not accept ownership or responsibility for the content or opinions expressed in any of the articles included in this issue. All articles are checked and tested before the release deadline is met but some faults may remain. The reader is responsible for all consequences, both to software and hardware, following the implementation of any of the advice or code printed. The MagPi does not claim to own any copyright licenses and all content of the articles are submitted with the responsibility lying with that of the article writer. This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit:

http://creativecommons.org/licenses/by-nc-sa/3.0/

Alternatively, send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.